# 第12章 関数ポインタ

## 対応C規格

* **主要対象:** C90
* **学習内容:** 関数ポインタの基本、コールバック関数、関数ポインタ配列、動的な関数呼び出し

## 学習目標

この章を完了すると、以下のことができるようになります：

* 関数ポインタの基本概念を理解する
* 関数ポインタの宣言と初期化ができる
* コールバック関数を実装できる
* 関数ポインタ配列を活用できる
* 動的な関数選択システムを作成できる

## 概要と詳細

### 関数ポインタとは？

関数ポインタは、関数のアドレスを格納する特殊なポインタです。これまでのポインタは変数のアドレスを扱いましたが、関数ポインタは関数のアドレスを扱います。

#### 日常生活での関数ポインタ

関数ポインタを理解するために、リモコンの例を考えてみましょう：

**テレビのリモコン** - ボタン1 → チャンネル1を表示する機能 - ボタン2 → チャンネル2を表示する機能 - ボタン3 → 音量を上げる機能

各ボタンは「どの機能を実行するか」を記憶しています。これが関数ポインタの概念です！

### なぜ関数ポインタが必要なのか？

1. **動的な関数選択**
   * 実行時に呼び出す関数を選べる
   * 条件によって処理を切り替えられる
2. **コールバック関数**
   * 処理の完了時に特定の関数を呼び出す
   * イベント駆動型プログラミングの基礎
3. **関数のテーブル化**
   * メニューシステムの実装
   * コマンドパターンの実現

### 関数ポインタの基本概念

関数ポインタは関数のアドレスを格納するポインタです。これにより、実行時に呼び出す関数を動的に決定できます。

#### 覚え方のコツ

通常のポインタ： int \*ptr; // intを指すポインタ  
関数ポインタ： int (\*ptr)(); // intを返す関数を指すポインタ

**重要**：関数ポインタでは括弧 () が必須です！

#### 関数ポインタの宣言

#include <stdio.h>  
  
/\* 通常の関数 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
int main(void)  
{  
 /\* 関数ポインタの宣言 \*/  
 int (\*operation)(int, int);  
   
 /\* 関数ポインタに関数のアドレスを代入 \*/  
 operation = add; /\* または &add \*/  
   
 /\* 関数ポインタを通じて関数を呼び出し \*/  
 printf("addition: %d\n", operation(10, 5));  
   
 /\* 別の関数を指すように変更 \*/  
 operation = subtract;  
 printf("subtraction: %d\n", operation(10, 5));  
   
 return 0;  
}

#### 関数ポインタの基本文法

#include <stdio.h>  
  
/\* さまざまな関数 \*/  
void greet(void)  
{  
 printf("こんにちは！\n");  
}  
  
int multiply(int x, int y)  
{  
 return x \* y;  
}  
  
double divide(double a, double b)  
{  
 if (b != 0.0) {  
 return a / b;  
 }  
 return 0.0;  
}  
  
int main(void)  
{  
 /\* さまざまな関数ポインタの宣言 \*/  
 void (\*greeting\_func)(void); /\* 引数なし、戻り値なし \*/  
 int (\*math\_func)(int, int); /\* int引数2つ、int戻り値 \*/  
 double (\*calc\_func)(double, double); /\* double引数2つ、double戻り値 \*/  
   
 /\* 関数ポインタの初期化 \*/  
 greeting\_func = greet;  
 math\_func = multiply;  
 calc\_func = divide;  
   
 /\* 関数ポインタを使った呼び出し \*/  
 greeting\_func(); /\* greet()を呼び出し \*/  
 printf("乗算: %d\n", math\_func(6, 7)); /\* multiply(6, 7)を呼び出し \*/  
 printf("除算: %.2f\n", calc\_func(10.0, 3.0)); /\* divide(10.0, 3.0)を呼び出し \*/  
   
 /\* 2つの異なる呼び出し方法 \*/  
 printf("直接呼び出し: %d\n", (\*math\_func)(8, 4)); /\* (\*ptr)(args) \*/  
 printf("間接呼び出し: %d\n", math\_func(8, 4)); /\* ptr(args) \*/  
   
 return 0;  
}

### 関数ポインタの実践的な使用

関数ポインタを実際のプログラムでどのように活用するか見ていきましょう。

#### なぜ実践が重要か？

関数ポインタは概念だけでは理解しにくいため、実際の使用例を通じて学ぶのが効果的です。

#### 計算機システム

#include <stdio.h>  
  
/\* 演算関数群 \*/  
double add\_op(double a, double b) { return a + b; }  
double sub\_op(double a, double b) { return a - b; }  
double mul\_op(double a, double b) { return a \* b; }  
double div\_op(double a, double b)   
{  
 if (b != 0.0) return a / b;  
 printf("エラー: ゼロ除算\n");  
 return 0.0;  
}  
  
/\* 演算を実行する関数 \*/  
double calculate(double a, double b, double (\*operation)(double, double))  
{  
 return operation(a, b);  
}  
  
/\* 演算子に基づいて関数を選択 \*/  
double (\*get\_operation(char op))(double, double)  
{  
 switch (op) {  
 case '+': return add\_op;  
 case '-': return sub\_op;  
 case '\*': return mul\_op;  
 case '/': return div\_op;  
 default: return NULL;  
 }  
}  
  
int main(void)  
{  
 double num1 = 20.0, num2 = 5.0;  
 char operators[] = {'+', '-', '\*', '/'};  
 const char\* op\_names[] = {"加算", "減算", "乗算", "除算"};  
 int i;  
   
 printf("数値1: %.1f, 数値2: %.1f\n\n", num1, num2);  
   
 /\* すべての演算を実行 \*/  
 for (i = 0; i < 4; i++)   
 {  
 double (\*op\_func)(double, double) = get\_operation(operators[i]);  
 if (op\_func != NULL)   
 {  
 double result = calculate(num1, num2, op\_func);  
 printf("%s (%c): %.2f\n", op\_names[i], operators[i], result);  
 }  
 }  
   
 return 0;  
}

#### 動的メニューシステム

#include <stdio.h>  
  
/\* メニュー項目の処理関数 \*/  
void show\_profile(void)  
{  
 printf("=== プロフィール表示 ===\n");  
 printf("名前: 田中太郎\n");  
 printf("年齢: 25歳\n");  
 printf("職業: エンジニア\n\n");  
}  
  
void show\_settings(void)  
{  
 printf("=== 設定画面 ===\n");  
 printf("言語: 日本語\n");  
 printf("テーマ: ダーク\n");  
 printf("通知: ON\n\n");  
}  
  
void show\_help(void)  
{  
 printf("=== ヘルプ ===\n");  
 printf("このアプリケーションの使用方法:\n");  
 printf("1. メニューから項目を選択\n");  
 printf("2. 処理が実行されます\n\n");  
}  
  
void exit\_app(void)  
{  
 printf("アプリケーションを終了します。\n");  
}  
  
/\* メニュー項目の構造体 \*/  
struct MenuItem {  
 char name[30];  
 void (\*handler)(void); /\* 関数ポインタ \*/  
};  
  
int main(void)  
{  
 /\* メニュー項目の定義 \*/  
 struct MenuItem menu[] = {  
 {"プロフィール", show\_profile},  
 {"設定", show\_settings},  
 {"ヘルプ", show\_help},  
 {"終了", exit\_app}  
 };  
   
 int menu\_size = sizeof(menu) / sizeof(menu[0]);  
 int choice;  
 int i;  
   
 do {  
 printf("=== メインメニュー ===\n");  
 for (i = 0; i < menu\_size; i++) {  
 printf("%d. %s\n", i + 1, menu[i].name);  
 }  
 printf("選択してください (1-%d): ", menu\_size);  
   
 if (scanf("%d", &choice) == 1 && choice >= 1 && choice <= menu\_size) {  
 printf("\n");  
 menu[choice - 1].handler(); /\* 関数ポインタを使って実行 \*/  
   
 if (choice == menu\_size) { /\* 終了が選択された場合 \*/  
 break;  
 }  
 } else {  
 printf("無効な選択です。\n\n");  
 /\* 入力バッファをクリア \*/  
 while (getchar() != '\n');  
 }  
 } while (1);  
   
 return 0;  
}

### コールバック関数

コールバック関数は他の関数に引数として渡される関数です。

#### コールバックの日常例

レストランでの注文を考えてみましょう：

1. 注文を受ける（関数呼び出し）
2. 料理を作る（処理）
3. **完成したら呼ぶ**（コールバック）

「完成したら何をするか」を事前に決めておくのがコールバックです！

#### 配列処理のコールバック

#include <stdio.h>  
  
/\* 配列の各要素に適用する関数群 \*/  
int square(int x) { return x \* x; }  
int cube(int x) { return x \* x \* x; }  
int double\_value(int x) { return x \* 2; }  
int increment(int x) { return x + 1; }  
  
/\* 配列の各要素に関数を適用 \*/  
void apply\_to\_array(int arr[], int size, int (\*func)(int))  
{  
 int i;  
 for (i = 0; i < size; i++) {  
 arr[i] = func(arr[i]);  
 }  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(int arr[], int size, const char\* label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[] = {1, 2, 3, 4, 5};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int temp[5];  
 int i;  
   
 print\_array(numbers, size, "元の配列");  
   
 /\* 2乗を適用 \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, square);  
 print\_array(temp, size, "2乗後");  
   
 /\* 3乗を適用 \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, cube);  
 print\_array(temp, size, "3乗後");  
   
 /\* 2倍を適用 \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, double\_value);  
 print\_array(temp, size, "2倍後");  
   
 /\* インクリメントを適用 \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, increment);  
 print\_array(temp, size, "インクリメント後");  
   
 return 0;  
}

#### ソートのコールバック（比較関数）

#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
/\* 比較関数の型定義 \*/  
typedef int (\*compare\_func\_t)(const void \*a, const void \*b);  
  
/\* 整数の比較関数 \*/  
int compare\_int\_asc(const void \*a, const void \*b)  
{  
 int ia = \*(const int\*)a;  
 int ib = \*(const int\*)b;  
 return (ia > ib) - (ia < ib); /\* 昇順 \*/  
}  
  
int compare\_int\_desc(const void \*a, const void \*b)  
{  
 int ia = \*(const int\*)a;  
 int ib = \*(const int\*)b;  
 return (ib > ia) - (ib < ia); /\* 降順 \*/  
}  
  
/\* 簡単なバブルソート（コールバック版） \*/  
void bubble\_sort(void \*base, size\_t num, size\_t size,   
 int (\*compare)(const void \*, const void \*))  
{  
 char \*arr = (char\*)base;  
 char \*temp = malloc(size);  
 size\_t i, j;  
   
 if (temp == NULL) return;  
   
 for (i = 0; i < num - 1; i++) {  
 for (j = 0; j < num - 1 - i; j++) {  
 void \*elem1 = arr + j \* size;  
 void \*elem2 = arr + (j + 1) \* size;  
   
 if (compare(elem1, elem2) > 0) {  
 /\* 要素を交換 \*/  
 memcpy(temp, elem1, size);  
 memcpy(elem1, elem2, size);  
 memcpy(elem2, temp, size);  
 }  
 }  
 }  
   
 free(temp);  
}  
  
/\* 配列を表示 \*/  
void print\_int\_array(int arr[], int size, const char\* label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[] = {42, 17, 56, 8, 23, 91, 3, 65};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int temp[8];  
 int i;  
   
 print\_int\_array(numbers, size, "元の配列");  
   
 /\* 昇順ソート \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 bubble\_sort(temp, size, sizeof(int), compare\_int\_asc);  
 print\_int\_array(temp, size, "昇順ソート");  
   
 /\* 降順ソート \*/  
 for (i = 0; i < size; i++) temp[i] = numbers[i];  
 bubble\_sort(temp, size, sizeof(int), compare\_int\_desc);  
 print\_int\_array(temp, size, "降順ソート");  
   
 return 0;  
}

### 関数ポインタ配列

関数ポインタの配列を使って、複数の関数を効率的に管理できます。

#### なぜ配列にするのか？

/\* 個別に管理（大変！） \*/  
int (\*func1)() = add;  
int (\*func2)() = sub;  
int (\*func3)() = mul;  
  
/\* 配列で管理（スッキリ！） \*/  
int (\*funcs[3])() = {add, sub, mul};

インデックスで関数を選択できるので、メニューシステムなどに最適です！

#### 演算関数配列

#include <stdio.h>  
  
/\* 演算関数群 \*/  
int add\_func(int a, int b) { return a + b; }  
int sub\_func(int a, int b) { return a - b; }  
int mul\_func(int a, int b) { return a \* b; }  
int div\_func(int a, int b) { return b != 0 ? a / b : 0; }  
int mod\_func(int a, int b) { return b != 0 ? a % b : 0; }  
  
int main(void)  
{  
 /\* 関数ポインタ配列の宣言と初期化 \*/  
 int (\*operations[])(int, int) =   
 {  
 add\_func, /\* インデックス0: 加算 \*/  
 sub\_func, /\* インデックス1: 減算 \*/  
 mul\_func, /\* インデックス2: 乗算 \*/  
 div\_func, /\* インデックス3: 除算 \*/  
 mod\_func /\* インデックス4: 剰余 \*/  
 };  
   
 const char\* op\_names[] = {"加算", "減算", "乗算", "除算", "剰余"};  
 const char\* op\_symbols[] = {"+", "-", "\*", "/", "%"};  
 int num\_ops = sizeof(operations) / sizeof(operations[0]);  
 int a = 20, b = 5;  
 int i;  
   
 printf("数値: a = %d, b = %d\n\n", a, b);  
   
 /\* すべての演算を実行 \*/  
 for (i = 0; i < num\_ops; i++)   
 {  
 int result = operations[i](a, b);  
 printf("%s (%s): %d %s %d = %d\n",   
 op\_names[i], op\_symbols[i], a, op\_symbols[i], b, result);  
 }  
   
 /\* 特定の演算だけ実行 \*/  
 printf("\n特定の演算:\n");  
 printf("乗算結果: %d\n", operations[2](a, b)); /\* mul\_func \*/  
 printf("除算結果: %d\n", operations[3](a, b)); /\* div\_func \*/  
   
 return 0;  
}

#### 状態機械（ステートマシン）

#include <stdio.h>  
#include <string.h>  
  
/\* 状態の定義 \*/  
typedef enum   
{  
 STATE\_IDLE,  
 STATE\_RUNNING,  
 STATE\_PAUSED,  
 STATE\_STOPPED,  
 STATE\_COUNT  
} State;  
  
/\* 状態処理関数の宣言 \*/  
void handle\_idle(void);  
void handle\_running(void);  
void handle\_paused(void);  
void handle\_stopped(void);  
  
/\* 現在の状態 \*/  
static State current\_state = STATE\_IDLE;  
  
/\* 状態処理関数配列 \*/  
void (\*state\_handlers[])(void) =   
{  
 handle\_idle, /\* STATE\_IDLE \*/  
 handle\_running, /\* STATE\_RUNNING \*/  
 handle\_paused, /\* STATE\_PAUSED \*/  
 handle\_stopped /\* STATE\_STOPPED \*/  
};  
  
const char\* state\_names[] =   
{  
 "待機中", "実行中", "一時停止", "停止中"  
};  
  
/\* 各状態の処理関数 \*/  
void handle\_idle(void)  
{  
 printf("[待機中] システムが待機状態です\n");  
 printf(" 利用可能なアクション: 開始(s)\n");  
}  
  
void handle\_running(void)  
{  
 printf("[実行中] システムが動作しています\n");  
 printf(" 利用可能なアクション: 一時停止(p), 停止(q)\n");  
}  
  
void handle\_paused(void)  
{  
 printf("[一時停止] システムが一時停止中です\n");  
 printf(" 利用可能なアクション: 再開(r), 停止(q)\n");  
}  
  
void handle\_stopped(void)  
{  
 printf("[停止中] システムが停止しました\n");  
 printf(" 利用可能なアクション: リセット(reset)\n");  
}  
  
/\* 状態遷移関数 \*/  
void transition\_to\_state(State new\_state)  
{  
 if (new\_state >= 0 && new\_state < STATE\_COUNT)   
 {  
 printf("状態遷移: %s -> %s\n",   
 state\_names[current\_state], state\_names[new\_state]);  
 current\_state = new\_state;  
 }  
}  
  
/\* 現在の状態を処理 \*/  
void process\_current\_state(void)  
{  
 if (current\_state >= 0 && current\_state < STATE\_COUNT)   
 {  
 state\_handlers[current\_state]();  
 }  
}  
  
int main(void)  
{  
 char command[10];  
   
 printf("=== ステートマシンデモ ===\n");  
 printf("コマンド: s(start), p(pause), r(resume), q(quit), reset\n\n");  
   
 while (1)   
 {  
 process\_current\_state();  
 printf("\nコマンドを入力してください: ");  
   
 if (scanf("%9s", command) == 1)   
 {  
 if (strcmp(command, "s") == 0 && current\_state == STATE\_IDLE)   
 {  
 transition\_to\_state(STATE\_RUNNING);  
 }  
 else if (strcmp(command, "p") == 0 && current\_state == STATE\_RUNNING)   
 {  
 transition\_to\_state(STATE\_PAUSED);  
 }  
 else if (strcmp(command, "r") == 0 && current\_state == STATE\_PAUSED)   
 {  
 transition\_to\_state(STATE\_RUNNING);  
 }  
 else if (strcmp(command, "q") == 0)   
 {  
 transition\_to\_state(STATE\_STOPPED);  
 }  
 else if (strcmp(command, "reset") == 0)   
 {  
 transition\_to\_state(STATE\_IDLE);  
 }  
 else if (strcmp(command, "exit") == 0)   
 {  
 printf("プログラムを終了します\n");  
 break;  
 }  
 else   
 {  
 printf("無効なコマンドまたは現在の状態では実行できません\n");  
 }  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

### 高度な関数ポインタ活用

より実践的な関数ポインタの活用方法を見ていきましょう。

#### プラグインシステム

プラグインシステムは、プログラムの機能を後から追加できる仕組みです。関数ポインタはこの実現に最適です！

#include <stdio.h>  
#include <string.h>  
  
/\* プラグインインターフェース \*/  
typedef struct   
{  
 char name[50];  
 char version[10];  
 void (\*init)(void);  
 void (\*execute)(void);  
 void (\*cleanup)(void);  
} Plugin;  
  
/\* プラグイン: ログ出力 \*/  
void log\_init(void) { printf("ログプラグイン初期化\n"); }  
void log\_execute(void) { printf("ログファイルに出力中...\n"); }  
void log\_cleanup(void) { printf("ログプラグイン終了処理\n"); }  
  
/\* プラグイン: データベース接続 \*/  
void db\_init(void) { printf("データベースプラグイン初期化\n"); }  
void db\_execute(void) { printf("データベースに接続中...\n"); }  
void db\_cleanup(void) { printf("データベース接続終了\n"); }  
  
/\* プラグイン: ネットワーク通信 \*/  
void net\_init(void) { printf("ネットワークプラグイン初期化\n"); }  
void net\_execute(void) { printf("ネットワーク通信実行中...\n"); }  
void net\_cleanup(void) { printf("ネットワーク接続終了\n"); }  
  
/\* プラグインの登録 \*/  
Plugin plugins[] =   
{  
 {"LogPlugin", "1.0", log\_init, log\_execute, log\_cleanup},  
 {"DatabasePlugin", "2.0", db\_init, db\_execute, db\_cleanup},  
 {"NetworkPlugin", "1.5", net\_init, net\_execute, net\_cleanup}  
};  
  
int plugin\_count = sizeof(plugins) / sizeof(plugins[0]);  
  
/\* プラグイン管理システム \*/  
void load\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン読み込み ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("プラグイン: %s (v%s)\n", plugins[i].name, plugins[i].version);  
 plugins[i].init();  
 }  
 printf("\n");  
}  
  
void execute\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン実行 ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("[%s] ", plugins[i].name);  
 plugins[i].execute();  
 }  
 printf("\n");  
}  
  
void unload\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン終了処理 ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("[%s] ", plugins[i].name);  
 plugins[i].cleanup();  
 }  
 printf("\n");  
}  
  
/\* 特定のプラグインを実行 \*/  
void execute\_plugin\_by\_name(const char\* name)  
{  
 int i;  
 for (i = 0; i < plugin\_count; i++)   
 {  
 if (strcmp(plugins[i].name, name) == 0)   
 {  
 printf("プラグイン '%s' を実行:\n", name);  
 plugins[i].execute();  
 return;  
 }  
 }  
 printf("プラグイン '%s' が見つかりません\n", name);  
}  
  
int main(void)  
{  
 load\_plugins();  
 execute\_plugins();  
   
 /\* 特定のプラグインのみ実行 \*/  
 execute\_plugin\_by\_name("DatabasePlugin");  
 execute\_plugin\_by\_name("InvalidPlugin");  
   
 unload\_plugins();  
   
 return 0;  
}

#### イベントハンドラシステム

#include <stdio.h>  
#include <string.h>  
  
/\* イベントタイプ \*/  
typedef enum   
{  
 EVENT\_CLICK,  
 EVENT\_KEYPRESS,  
 EVENT\_MOUSE\_MOVE,  
 EVENT\_WINDOW\_CLOSE,  
 EVENT\_TYPE\_COUNT  
} EventType;  
  
/\* イベントデータ \*/  
typedef struct   
{  
 EventType type;  
 int x, y; /\* 座標 \*/  
 int key\_code; /\* キーコード \*/  
 char message[100];  
} Event;  
  
/\* イベントハンドラ関数の型 \*/  
typedef void (\*EventHandler)(const Event\* event);  
  
/\* イベントハンドラ配列 \*/  
EventHandler event\_handlers[EVENT\_TYPE\_COUNT] = {NULL};  
  
/\* 各イベントのハンドラ実装 \*/  
void handle\_click(const Event\* event)  
{  
 printf("クリックイベント: 座標(%d, %d) - %s\n",   
 event->x, event->y, event->message);  
}  
  
void handle\_keypress(const Event\* event)  
{  
 printf("キー押下イベント: キーコード %d - %s\n",   
 event->key\_code, event->message);  
}  
  
void handle\_mouse\_move(const Event\* event)  
{  
 printf("マウス移動イベント: 座標(%d, %d)\n", event->x, event->y);  
}  
  
void handle\_window\_close(const Event\* event)  
{  
 printf("ウィンドウ閉じるイベント: %s\n", event->message);  
}  
  
/\* イベントハンドラの登録 \*/  
void register\_event\_handler(EventType type, EventHandler handler)  
{  
 if (type >= 0 && type < EVENT\_TYPE\_COUNT)   
 {  
 event\_handlers[type] = handler;  
 printf("イベントハンドラ登録: タイプ %d\n", type);  
 }  
}  
  
/\* イベントの処理 \*/  
void process\_event(const Event\* event)  
{  
 if (event->type >= 0 && event->type < EVENT\_TYPE\_COUNT &&   
 event\_handlers[event->type] != NULL)   
 {  
 event\_handlers[event->type](event);  
 }   
 else   
 {  
 printf("未処理のイベント: タイプ %d\n", event->type);  
 }  
}  
  
/\* イベントの作成補助関数 \*/  
Event create\_click\_event(int x, int y, const char\* msg)  
{  
 Event event = {EVENT\_CLICK, x, y, 0, ""};  
 strncpy(event.message, msg, sizeof(event.message) - 1);  
 return event;  
}  
  
Event create\_keypress\_event(int key\_code, const char\* msg)  
{  
 Event event = {EVENT\_KEYPRESS, 0, 0, key\_code, ""};  
 strncpy(event.message, msg, sizeof(event.message) - 1);  
 return event;  
}  
  
int main(void)  
{  
 printf("=== イベントハンドラシステム ===\n\n");  
   
 /\* イベントハンドラを登録 \*/  
 register\_event\_handler(EVENT\_CLICK, handle\_click);  
 register\_event\_handler(EVENT\_KEYPRESS, handle\_keypress);  
 register\_event\_handler(EVENT\_MOUSE\_MOVE, handle\_mouse\_move);  
 register\_event\_handler(EVENT\_WINDOW\_CLOSE, handle\_window\_close);  
   
 printf("\n=== イベント処理テスト ===\n");  
   
 /\* さまざまなイベントを生成・処理 \*/  
 Event click\_event = create\_click\_event(100, 200, "ボタンクリック");  
 process\_event(&click\_event);  
   
 Event key\_event = create\_keypress\_event(65, "Aキー押下");  
 process\_event(&key\_event);  
   
 Event mouse\_event = {EVENT\_MOUSE\_MOVE, 150, 300, 0, ""};  
 process\_event(&mouse\_event);  
   
 Event close\_event = {EVENT\_WINDOW\_CLOSE, 0, 0, 0, "アプリケーション終了"};  
 process\_event(&close\_event);  
   
 /\* 未登録のイベントタイプ \*/  
 Event unknown\_event = {99, 0, 0, 0, ""};  
 process\_event(&unknown\_event);  
   
 return 0;  
}

### 関数ポインタのtypedef

複雑な関数ポインタの型を簡潔に書くためにtypedefを使用します。

#### typedefを使うメリット

/\* typedefなし（読みにくい！） \*/  
void sort(int arr[], int size, int (\*compare)(int, int));  
  
/\* typedefあり（スッキリ！） \*/  
typedef int (\*CompareFunc)(int, int);  
void sort(int arr[], int size, CompareFunc compare);

特に複雑な関数ポインタでは、typedefの効果が絶大です！

#include <stdio.h>  
  
/\* 関数ポインタの型定義 \*/  
typedef int (\*BinaryOperation)(int, int);  
typedef void (\*EventCallback)(int event\_id, const char\* message);  
typedef double (\*MathFunction)(double);  
  
/\* 演算関数 \*/  
int add\_nums(int a, int b) { return a + b; }  
int mul\_nums(int a, int b) { return a \* b; }  
  
/\* イベントコールバック関数 \*/  
void on\_start(int id, const char\* msg)   
{  
 printf("開始イベント[%d]: %s\n", id, msg);  
}  
  
void on\_stop(int id, const char\* msg)   
{  
 printf("停止イベント[%d]: %s\n", id, msg);  
}  
  
/\* 数学関数 \*/  
double square\_root(double x) { return x \* x; } /\* 簡易版 \*/  
double absolute(double x) { return x < 0 ? -x : x; }  
  
/\* 関数を実行する汎用関数 \*/  
int execute\_binary\_op(int a, int b, BinaryOperation op)  
{  
 return op(a, b);  
}  
  
void trigger\_event(int id, const char\* message, EventCallback callback)  
{  
 callback(id, message);  
}  
  
double apply\_math\_func(double value, MathFunction func)  
{  
 return func(value);  
}  
  
int main(void)  
{  
 /\* 型定義を使った関数ポインタの使用 \*/  
 BinaryOperation math\_op;  
 EventCallback event\_handler;  
 MathFunction math\_func;  
   
 printf("=== typedef を使った関数ポインタ ===\n\n");  
   
 /\* 数値演算 \*/  
 math\_op = add\_nums;  
 printf("加算: %d\n", execute\_binary\_op(10, 5, math\_op));  
   
 math\_op = mul\_nums;  
 printf("乗算: %d\n", execute\_binary\_op(10, 5, math\_op));  
   
 /\* イベント処理 \*/  
 event\_handler = on\_start;  
 trigger\_event(1, "システム開始", event\_handler);  
   
 event\_handler = on\_stop;  
 trigger\_event(2, "システム終了", event\_handler);  
   
 /\* 数学関数 \*/  
 math\_func = square\_root;  
 printf("二乗: %.2f\n", apply\_math\_func(5.0, math\_func));  
   
 math\_func = absolute;  
 printf("絶対値: %.2f\n", apply\_math\_func(-7.5, math\_func));  
   
 return 0;  
}

## サンプルコード

### 基本的な関数ポインタ

プログラムファイル: examples/function\_pointer\_basics.c

関数ポインタの宣言、初期化、呼び出しの基本を学習します。

### コールバック関数の実装

プログラムファイル: examples/callback\_functions.c

コールバック関数を使った配列処理やイベント処理を学習します。

### 関数ポインタ配列の活用

プログラムファイル: examples/function\_pointer\_arrays.c

関数ポインタ配列を使った動的な関数選択システムを学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic function\_pointer\_basics.c -o function\_pointer\_basics  
  
# 実行  
./function\_pointer\_basics

## 演習課題

### 基礎問題

1. **簡単な計算機**
   * 関数ポインタを使って四則演算ができる計算機を作成してください
2. **文字列処理関数**
   * 文字列を大文字・小文字・反転する関数を作成し、関数ポインタで切り替えるプログラムを作成してください
3. **配列フィルタ**
   * 配列から条件に合う要素だけを抽出する関数を、コールバック関数を使って実装してください

### 応用問題

1. **ソート関数**
   * 比較関数を関数ポインタで受け取る汎用ソート関数を実装してください
2. **コマンドシステム**
   * コマンドとそれに対応する処理関数をマッピングするシステムを作成してください
3. **イベントディスパッチャ**
   * 複数のイベントタイプとハンドラを管理するイベントシステムを実装してください

### 発展問題

1. **プラグインアーキテクチャ**
   * 動的にプラグインを追加・削除できるシステムを作成してください
2. **有限状態機械**
   * 関数ポインタ配列を使った本格的な状態機械を実装してください
3. **多層コールバックシステム**
   * コールバック関数が別のコールバックを呼ぶような多層システムを作成してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make function\_pointer\_basics  
  
# bin/ディレクトリ内の実行ファイルを実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 関数ポインタの初期化は静的初期化のみ
* インライン関数との組み合わせは不可
* 可変引数関数ポインタは制限的

### C99以降の拡張

* インライン関数ポインタのサポート
* 複合リテラルでの関数ポインタ初期化
* より柔軟な初期化構文

## よくある間違い

### 1. 括弧の位置

/\* NG: 関数を返すポインタになってしまう \*/  
int \*func(int, int); /\* これは関数宣言 \*/  
  
/\* OK: 関数へのポインタ \*/  
int (\*func)(int, int); /\* 括弧が重要！ \*/

### 2. 型の不一致

/\* NG: 引数の型が違う \*/  
void func(int x) { }  
void (\*ptr)(double) = func; /\* エラー！ \*/  
  
/\* OK: 型を一致させる \*/  
void (\*ptr)(int) = func; /\* OK \*/

### 3. NULLチェック忘れ

/\* NG: NULLポインタの呼び出し \*/  
void (\*func\_ptr)(void) = NULL;  
func\_ptr(); /\* セグメンテーション違反 \*/  
  
/\* OK: NULLチェック \*/  
if (func\_ptr != NULL) {  
 func\_ptr();  
}

### 4. 配列の初期化エラー

/\* NG: サイズ不一致 \*/  
void (\*funcs[3])(void) = {func1, func2}; /\* 3番目が未初期化 \*/  
  
/\* OK: 完全な初期化またはサイズ自動決定 \*/  
void (\*funcs[])(void) = {func1, func2}; /\* サイズ2の配列 \*/

## 次の章へ

関数ポインタを理解したら、[高度なトピック](../advanced/README.md) に進んでください。

## 参考資料

* [C言語関数ポインタリファレンス](https://ja.cppreference.com/w/c/language/pointer#.E9.96.A2.E6.95.B0.E3.83.9D.E3.82.A4.E3.83.B3.E3.82.BF)
* [コールバック関数の詳細](https://ja.cppreference.com/w/c/algorithm/qsort)
* [関数ポインタの高度な使用法](https://ja.cppreference.com/w/c/language/function_definition)

# サンプルコード

## callback\_functions.c

/\* コールバック関数の活用 \*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/\* 配列処理用のコールバック関数群 \*/  
int square(int x) { return x \* x; }  
int cube(int x) { return x \* x \* x; }  
int double\_value(int x) { return x \* 2; }  
int increment(int x) { return x + 1; }  
  
/\* 条件判定用のコールバック関数群 \*/  
int is\_even(int x) { return x % 2 == 0; }  
int is\_odd(int x) { return x % 2 != 0; }  
int is\_positive(int x) { return x > 0; }  
int is\_negative(int x) { return x < 0; }  
  
/\* 配列の各要素に関数を適用 \*/  
void apply\_to\_array(int arr[], int size, int (\*func)(int))  
{  
 int i;  
 for (i = 0; i < size; i++)  
 {  
 arr[i] = func(arr[i]);  
 }  
}  
  
/\* 配列をフィルタリング \*/  
int filter\_array(int source[], int dest[], int size, int (\*predicate)(int))  
{  
 int count = 0;  
 int i;  
  
 for (i = 0; i < size; i++)  
 {  
 if (predicate(source[i]))  
 {  
 dest[count++] = source[i];  
 }  
 }  
  
 return count;  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(int arr[], int size, const char \*label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++)  
 {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* 文字列処理用のコールバック関数群 \*/  
void to\_upper\_char(char \*c)  
{  
 if (\*c >= 'a' && \*c <= 'z')  
 {  
 \*c = \*c - 'a' + 'A';  
 }  
}  
  
void to\_lower\_char(char \*c)  
{  
 if (\*c >= 'A' && \*c <= 'Z')  
 {  
 \*c = \*c - 'A' + 'a';  
 }  
}  
  
void replace\_space\_with\_underscore(char \*c)  
{  
 if (\*c == ' ')  
 {  
 \*c = '\_';  
 }  
}  
  
/\* 文字列の各文字にコールバック関数を適用 \*/  
void process\_string(char str[], void (\*processor)(char \*))  
{  
 int i;  
 for (i = 0; str[i] != '\0'; i++)  
 {  
 processor(&str[i]);  
 }  
}  
  
/\* ソート用の比較関数 \*/  
int compare\_ascending(int a, int b)  
{  
 return a - b;  
}  
  
int compare\_descending(int a, int b)  
{  
 return b - a;  
}  
  
/\* バブルソート（コールバック版） \*/  
void bubble\_sort(int arr[], int size, int (\*compare)(int, int))  
{  
 int i, j, temp;  
  
 for (i = 0; i < size - 1; i++)  
 {  
 for (j = 0; j < size - 1 - i; j++)  
 {  
 if (compare(arr[j], arr[j + 1]) > 0)  
 {  
 temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
}  
  
/\* イベントハンドラシステム \*/  
void on\_start\_event(const char \*message)  
{  
 printf("[開始イベント] %s\n", message);  
}  
  
void on\_stop\_event(const char \*message)  
{  
 printf("[停止イベント] %s\n", message);  
}  
  
void on\_error\_event(const char \*message)  
{  
 printf("[エラーイベント] %s\n", message);  
}  
  
/\* イベントを処理する関数 \*/  
void trigger\_event(const char \*message, void (\*event\_handler)(const char \*))  
{  
 if (event\_handler != NULL)  
 {  
 event\_handler(message);  
 }  
}  
  
/\* 計算処理のログ付き実行 \*/  
int execute\_with\_logging(int a, int b, int (\*operation)(int, int), const char \*op\_name)  
{  
 printf("計算開始: %s(%d, %d)\n", op\_name, a, b);  
 int result = operation(a, b);  
 printf("計算完了: %s(%d, %d) = %d\n", op\_name, a, b, result);  
 return result;  
}  
  
/\* 簡単な加算関数（ログ用） \*/  
int add\_for\_logging(int a, int b)  
{  
 return a + b;  
}  
  
int multiply\_for\_logging(int a, int b)  
{  
 return a \* b;  
}  
  
int main(void)  
{  
 printf("=== コールバック関数の活用 ===\n\n");  
  
 /\* 1. 配列処理のコールバック \*/  
 printf("=== 配列処理のコールバック ===\n");  
 int numbers[] = {1, 2, 3, 4, 5};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int temp[5];  
 int i;  
  
 print\_array(numbers, size, "元の配列");  
  
 /\* 2乗を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, square);  
 print\_array(temp, size, "2乗後");  
  
 /\* 3乗を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, cube);  
 print\_array(temp, size, "3乗後");  
  
 /\* 2倍を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, double\_value);  
 print\_array(temp, size, "2倍後");  
  
 /\* 2. 配列フィルタリング \*/  
 printf("\n=== 配列フィルタリング ===\n");  
 int test\_array[] = {-3, -1, 0, 2, 4, 5, 7, 8};  
 int test\_size = sizeof(test\_array) / sizeof(test\_array[0]);  
 int filtered[8];  
 int filtered\_count;  
  
 print\_array(test\_array, test\_size, "元の配列");  
  
 /\* 偶数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_even);  
 print\_array(filtered, filtered\_count, "偶数のみ");  
  
 /\* 奇数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_odd);  
 print\_array(filtered, filtered\_count, "奇数のみ");  
  
 /\* 正の数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_positive);  
 print\_array(filtered, filtered\_count, "正の数のみ");  
  
 /\* 3. 文字列処理のコールバック \*/  
 printf("\n=== 文字列処理のコールバック ===\n");  
 char text1[] = "Hello World";  
 char text2[] = "Hello World";  
 char text3[] = "Hello World";  
  
 printf("元の文字列: '%s'\n", text1);  
  
 process\_string(text1, to\_upper\_char);  
 printf("大文字変換: '%s'\n", text1);  
  
 process\_string(text2, to\_lower\_char);  
 printf("小文字変換: '%s'\n", text2);  
  
 process\_string(text3, replace\_space\_with\_underscore);  
 printf("スペース置換: '%s'\n", text3);  
  
 /\* 4. ソートのコールバック \*/  
 printf("\n=== ソートのコールバック ===\n");  
 int sort\_array[] = {64, 34, 25, 12, 22, 11, 90, 5};  
 int sort\_size = sizeof(sort\_array) / sizeof(sort\_array[0]);  
 int sort\_temp[8];  
  
 print\_array(sort\_array, sort\_size, "元の配列");  
  
 /\* 昇順ソート \*/  
 for (i = 0; i < sort\_size; i++)  
 sort\_temp[i] = sort\_array[i];  
 bubble\_sort(sort\_temp, sort\_size, compare\_ascending);  
 print\_array(sort\_temp, sort\_size, "昇順ソート");  
  
 /\* 降順ソート \*/  
 for (i = 0; i < sort\_size; i++)  
 sort\_temp[i] = sort\_array[i];  
 bubble\_sort(sort\_temp, sort\_size, compare\_descending);  
 print\_array(sort\_temp, sort\_size, "降順ソート");  
  
 /\* 5. イベントハンドラシステム \*/  
 printf("\n=== イベントハンドラシステム ===\n");  
  
 trigger\_event("アプリケーション開始", on\_start\_event);  
 trigger\_event("処理中にエラーが発生", on\_error\_event);  
 trigger\_event("アプリケーション終了", on\_stop\_event);  
  
 /\* 6. ログ付き計算実行 \*/  
 printf("\n=== ログ付き計算実行 ===\n");  
  
 execute\_with\_logging(10, 5, add\_for\_logging, "加算");  
 execute\_with\_logging(7, 8, multiply\_for\_logging, "乗算");  
  
 printf("\n=== コールバック関数デモ完了 ===\n");  
  
 return 0;  
}

## function\_pointer\_arrays.c

/\* 関数ポインタ配列の活用 \*/  
#include <stdio.h>  
#include <string.h>  
  
/\* 計算機システム用の演算関数群 \*/  
double add\_operation(double a, double b) { return a + b; }  
double sub\_operation(double a, double b) { return a - b; }  
double mul\_operation(double a, double b) { return a \* b; }  
double div\_operation(double a, double b)  
{  
 if (b != 0.0)  
 return a / b;  
 printf("エラー: ゼロ除算\n");  
 return 0.0;  
}  
  
/\* メニューシステム用の処理関数群 \*/  
void show\_profile(void)  
{  
 printf("=== プロフィール表示 ===\n");  
 printf("名前: 田中太郎\n");  
 printf("年齢: 25歳\n");  
 printf("職業: エンジニア\n");  
 printf("========================\n");  
}  
  
void show\_settings(void)  
{  
 printf("=== 設定画面 ===\n");  
 printf("言語: 日本語\n");  
 printf("テーマ: ダーク\n");  
 printf("通知: ON\n");  
 printf("===============\n");  
}  
  
void show\_help(void)  
{  
 printf("=== ヘルプ ===\n");  
 printf("このアプリケーションの使用方法:\n");  
 printf("1. メニューから項目を選択\n");  
 printf("2. 処理が実行されます\n");  
 printf("3. メインメニューに戻ります\n");  
 printf("=============\n");  
}  
  
void show\_about(void)  
{  
 printf("=== このアプリについて ===\n");  
 printf("関数ポインタ配列デモ v1.0\n");  
 printf("C言語学習用サンプルプログラム\n");  
 printf("========================\n");  
}  
  
void exit\_application(void)  
{  
 printf("アプリケーションを終了します。\n");  
}  
  
/\* ゲーム状態管理用の関数群 \*/  
void state\_menu(void)  
{  
 printf("[メニュー状態] ゲームメニューを表示中\n");  
 printf(" 利用可能なアクション: 開始(s), 設定(c), 終了(q)\n");  
}  
  
void state\_playing(void)  
{  
 printf("[プレイ中状態] ゲームをプレイ中\n");  
 printf(" 利用可能なアクション: 一時停止(p), メニューに戻る(m)\n");  
}  
  
void state\_paused(void)  
{  
 printf("[一時停止状態] ゲームが一時停止中\n");  
 printf(" 利用可能なアクション: 再開(r), メニューに戻る(m)\n");  
}  
  
void state\_settings(void)  
{  
 printf("[設定状態] ゲーム設定を調整中\n");  
 printf(" 利用可能なアクション: 保存(s), キャンセル(c)\n");  
}  
  
/\* データ変換関数群 \*/  
int to\_upper\_case(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return c - 'a' + 'A';  
 }  
 return c;  
}  
  
int to\_lower\_case(int c)  
{  
 if (c >= 'A' && c <= 'Z')  
 {  
 return c - 'A' + 'a';  
 }  
 return c;  
}  
  
int rot13\_transform(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return (c - 'a' + 13) % 26 + 'a';  
 }  
 if (c >= 'A' && c <= 'Z')  
 {  
 return (c - 'A' + 13) % 26 + 'A';  
 }  
 return c;  
}  
  
int reverse\_case(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return c - 'a' + 'A';  
 }  
 if (c >= 'A' && c <= 'Z')  
 {  
 return c - 'A' + 'a';  
 }  
 return c;  
}  
  
/\* メニュー項目構造体 \*/  
struct MenuItem  
{  
 char name[30];  
 void (\*handler)(void);  
};  
  
/\* ゲーム状態の列挙型 \*/  
typedef enum  
{  
 GAME\_MENU = 0,  
 GAME\_PLAYING = 1,  
 GAME\_PAUSED = 2,  
 GAME\_SETTINGS = 3,  
 GAME\_STATE\_COUNT = 4  
} GameState;  
  
/\* 文字変換関数の適用 \*/  
void apply\_transform(char \*str, int (\*transform)(int))  
{  
 int i;  
 for (i = 0; str[i] != '\0'; i++)  
 {  
 str[i] = transform(str[i]);  
 }  
}  
  
/\* 配列の表示 \*/  
void print\_double\_array(double arr[], int size, const char \*label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++)  
 {  
 printf("%.2f ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 printf("=== 関数ポインタ配列の活用 ===\n\n");  
  
 /\* 1. 計算機システム \*/  
 printf("=== 計算機システム ===\n");  
  
 /\* 関数ポインタ配列の定義 \*/  
 double (\*operations[])(double, double) = {  
 add\_operation, /\* インデックス0: 加算 \*/  
 sub\_operation, /\* インデックス1: 減算 \*/  
 mul\_operation, /\* インデックス2: 乗算 \*/  
 div\_operation /\* インデックス3: 除算 \*/  
 };  
  
 const char \*op\_names[] = {"加算", "減算", "乗算", "除算"};  
 const char \*op\_symbols[] = {"+", "-", "\*", "/"};  
 int num\_ops = sizeof(operations) / sizeof(operations[0]);  
 double a = 15.0, b = 4.0;  
 int i;  
  
 printf("数値: a = %.1f, b = %.1f\n", a, b);  
  
 /\* 全ての演算を実行 \*/  
 for (i = 0; i < num\_ops; i++)  
 {  
 double result = operations[i](a, b);  
 printf("%s (%s): %.1f %s %.1f = %.2f\n",  
 op\_names[i], op\_symbols[i], a, op\_symbols[i], b, result);  
 }  
  
 /\* 特定の演算のみ実行 \*/  
 printf("\n特定の演算:\n");  
 printf("乗算結果: %.2f\n", operations[2](a, b)); /\* mul\_operation \*/  
 printf("除算結果: %.2f\n", operations[3](a, b)); /\* div\_operation \*/  
  
 /\* 2. メニューシステム \*/  
 printf("\n=== メニューシステム ===\n");  
  
 /\* メニュー項目の定義 \*/  
 struct MenuItem menu[] = {  
 {"プロフィール", show\_profile},  
 {"設定", show\_settings},  
 {"ヘルプ", show\_help},  
 {"このアプリについて", show\_about},  
 {"終了", exit\_application}};  
  
 int menu\_size = sizeof(menu) / sizeof(menu[0]);  
  
 printf("利用可能なメニュー項目:\n");  
 for (i = 0; i < menu\_size; i++)  
 {  
 printf("%d. %s\n", i + 1, menu[i].name);  
 }  
  
 /\* メニュー項目の実行例 \*/  
 printf("\nメニュー実行例:\n");  
 menu[0].handler(); /\* プロフィール表示 \*/  
 menu[2].handler(); /\* ヘルプ表示 \*/  
  
 /\* 3. ゲーム状態管理システム \*/  
 printf("\n=== ゲーム状態管理システム ===\n");  
  
 /\* 状態処理関数配列 \*/  
 void (\*state\_handlers[])(void) = {  
 state\_menu, /\* GAME\_MENU \*/  
 state\_playing, /\* GAME\_PLAYING \*/  
 state\_paused, /\* GAME\_PAUSED \*/  
 state\_settings /\* GAME\_SETTINGS \*/  
 };  
  
 const char \*state\_names[] = {  
 "メニュー", "プレイ中", "一時停止", "設定"};  
  
 GameState current\_state;  
  
 printf("各ゲーム状態の処理:\n");  
 for (current\_state = GAME\_MENU; current\_state < GAME\_STATE\_COUNT; current\_state++)  
 {  
 printf("\n現在の状態: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
 }  
  
 /\* 4. 動的な状態遷移 \*/  
 printf("\n=== 動的な状態遷移 ===\n");  
  
 current\_state = GAME\_MENU;  
 printf("初期状態: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 /\* 状態遷移の例 \*/  
 current\_state = GAME\_PLAYING;  
 printf("\n状態遷移: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 current\_state = GAME\_PAUSED;  
 printf("\n状態遷移: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 /\* 5. 文字変換システム \*/  
 printf("\n=== 文字変換システム ===\n");  
  
 /\* 変換関数配列 \*/  
 int (\*transforms[])(int) = {  
 to\_upper\_case,  
 to\_lower\_case,  
 rot13\_transform,  
 reverse\_case};  
  
 const char \*transform\_names[] = {  
 "大文字変換",  
 "小文字変換",  
 "ROT13暗号",  
 "大小文字反転"};  
  
 int num\_transforms = sizeof(transforms) / sizeof(transforms[0]);  
 char original[] = "Hello World!";  
 char test\_str[50];  
  
 printf("元の文字列: '%s'\n", original);  
  
 /\* 各変換を適用 \*/  
 for (i = 0; i < num\_transforms; i++)  
 {  
 strcpy(test\_str, original);  
 apply\_transform(test\_str, transforms[i]);  
 printf("%s: '%s'\n", transform\_names[i], test\_str);  
 }  
  
 /\* 6. 複数の数値に同じ演算を適用 \*/  
 printf("\n=== 複数数値への演算適用 ===\n");  
  
 double numbers[] = {1.0, 2.0, 3.0, 4.0, 5.0};  
 double results[5];  
 int numbers\_size = sizeof(numbers) / sizeof(numbers[0]);  
  
 print\_double\_array(numbers, numbers\_size, "元の配列");  
  
 /\* 各演算を全ての数値ペア (numbers[i], 2.0) に適用 \*/  
 for (i = 0; i < num\_ops; i++)  
 {  
 int j;  
 for (j = 0; j < numbers\_size; j++)  
 {  
 results[j] = operations[i](numbers[j], 2.0);  
 }  
 printf("%s (x, 2.0): ", op\_names[i]);  
 print\_double\_array(results, numbers\_size, "結果");  
 }  
  
 /\* 7. 条件付き関数実行 \*/  
 printf("\n=== 条件付き関数実行 ===\n");  
  
 printf("演算インデックス指定実行:\n");  
 int op\_index = 2; /\* 乗算 \*/  
 if (op\_index >= 0 && op\_index < num\_ops)  
 {  
 double result = operations[op\_index](10.0, 3.0);  
 printf("%s: 10.0 %s 3.0 = %.2f\n",  
 op\_names[op\_index], op\_symbols[op\_index], result);  
 }  
  
 /\* 複数のメニュー項目を一度に実行 \*/  
 printf("\n複数メニュー項目の実行:\n");  
 int menu\_indices[] = {1, 3}; /\* 設定, このアプリについて \*/  
 int indices\_count = sizeof(menu\_indices) / sizeof(menu\_indices[0]);  
  
 for (i = 0; i < indices\_count; i++)  
 {  
 int idx = menu\_indices[i];  
 if (idx >= 0 && idx < menu\_size)  
 {  
 printf("\n[%s] 実行:\n", menu[idx].name);  
 menu[idx].handler();  
 }  
 }  
  
 printf("\n=== 関数ポインタ配列デモ完了 ===\n");  
  
 return 0;  
}

## function\_pointer\_basic.c

/\* 関数ポインタの基本 \*/  
#include <stdio.h>  
  
/\* 数学演算関数群 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
int multiply(int a, int b)  
{  
 return a \* b;  
}  
  
int divide(int a, int b)  
{  
 if (b != 0)  
 {  
 return a / b;  
 }  
 printf("エラー: ゼロ除算\n");  
 return 0;  
}  
  
/\* 関数ポインタを使った計算 \*/  
int calculate(int x, int y, int (\*operation)(int, int))  
{  
 return operation(x, y);  
}  
  
/\* 演算子に基づいて関数を選択 \*/  
int (\*get\_operation(char op))(int, int)  
{  
 switch (op)  
 {  
 case '+':  
 return add;  
 case '-':  
 return subtract;  
 case '\*':  
 return multiply;  
 case '/':  
 return divide;  
 default:  
 return NULL;  
 }  
}  
  
int main(void)  
{  
 printf("=== 関数ポインタの基本 ===\n\n");  
  
 /\* 基本的な関数ポインタの使用 \*/  
 int (\*math\_func)(int, int);  
 int a = 10, b = 5;  
  
 printf("数値: a = %d, b = %d\n\n", a, b);  
  
 /\* 加算 \*/  
 math\_func = add;  
 printf("加算: %d + %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 減算 \*/  
 math\_func = subtract;  
 printf("減算: %d - %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 乗算 \*/  
 math\_func = multiply;  
 printf("乗算: %d \* %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 除算 \*/  
 math\_func = divide;  
 printf("除算: %d / %d = %d\n", a, b, math\_func(a, b));  
  
 printf("\n=== 関数選択システム ===\n");  
  
 /\* 演算子による関数選択 \*/  
 char operators[] = {'+', '-', '\*', '/', '%'};  
 const char \*op\_names[] = {"加算", "減算", "乗算", "除算", "不明"};  
 int i;  
  
 for (i = 0; i < 5; i++)  
 {  
 int (\*op\_func)(int, int) = get\_operation(operators[i]);  
 if (op\_func != NULL)  
 {  
 int result = calculate(a, b, op\_func);  
 printf("%s (%c): %d\n", op\_names[i], operators[i], result);  
 }  
 else  
 {  
 printf("%s (%c): サポートされていません\n", op\_names[i], operators[i]);  
 }  
 }  
  
 printf("\n=== 動的な演算実行 ===\n");  
  
 /\* 対話的な演算実行 \*/  
 char op;  
 printf("演算子を入力してください (+, -, \*, /): ");  
 if (scanf(" %c", &op) == 1)  
 {  
 int (\*selected\_op)(int, int) = get\_operation(op);  
 if (selected\_op != NULL)  
 {  
 int result = calculate(15, 3, selected\_op);  
 printf("15 %c 3 = %d\n", op, result);  
 }  
 else  
 {  
 printf("無効な演算子です\n");  
 }  
 }  
  
 return 0;  
}